



Docket No. 1073.060

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

RECEIVED

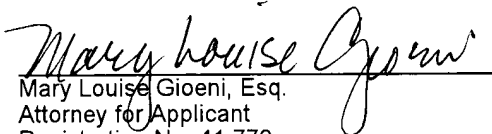
SEP 24 2001

TECH CENTER 1600/2900

Applicant: Diller et al. Docket: 1073.060
Serial No.: 09/595,096 Group Art Unit: 1631
Filed: 06/15/2000 Examiner: M. Sheinberg
Title: MOLECULAR DOCKING TECHNIQUE FOR SCREENING OF
COMBINATORIAL LIBRARIES

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the U.S. Postal Service as first class mail in an envelope addressed to: Assistant Commissioner for Patent and Trademarks, on September 17, 2001.


Mary Louise Gioeni, Esq.
Attorney for Applicant
Registration No. 41,779

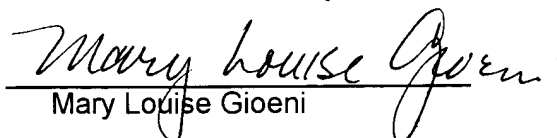
Date of Signature: September 17, 2001

To: Assistant Commissioner for Patents
Washington, D.C. 20231

Declaration Regarding Material Incorporated by Reference

1. I, Mary Louise Gioeni, am a practitioner representing the Applicants for the above-designated patent application.
2. The disclosure of the application has been amended to include material incorporated by reference.
3. The amendatory material consists of the same material incorporated by reference in the referencing application.
4. A copy of the relevant pages from the referenced publication is enclosed.
5. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Date: September 17, 2001

By: 
Mary Louise Gioeni

*fret=dbrent(ax,xx,bx,fldim,dfldim,TOL,&xmin);
 for (j=1;j<=n;j++) { Construct the vector results to return.
 xi[j] *= xmin;
 p[j] += xi[j];
 }
 free_vector(xicom,1,n);
 free_vector(pcom,1,n);

#include "nrutil.h"

extern int ncom; Defined in dlinmin.
 extern float *pcom,*xicom,(*nrfunc)(float []);
 extern void (*nrdfun)(float [], float []);

float dfldim(float x)
 {
 int j;
 float df1=0.0;
 float *xt,*df;

 xt=vector(1,ncom);
 df=vector(1,ncom);
 for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
 (*nrdfun)(xt,df);
 for (j=1;j<=ncom;j++) df1 += df[j]*xicom[j];
 free_vector(df,1,ncom);
 free_vector(xt,1,ncom);
 return df1;
 }

CITED REFERENCES AND FURTHER READING:

- Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), §2.3. [1]
 Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.7 (by K.W. Brodlie). [2]
 Stoer, J. and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §8.7.

10.7 Variable Metric Methods in Multidimensions

The goal of *variable metric* methods, which are sometimes called *quasi-Newton* methods, is not different from the goal of conjugate gradient methods: to accumulate information from successive line minimizations so that N such line minimizations lead to the exact minimum of a quadratic form in N dimensions. In that case, the method will also be quadratically convergent for more general smooth functions.

Both variable metric and conjugate gradient methods require that you are able to compute your function's gradient, or first partial derivatives, at arbitrary points. The variable metric approach differs from the conjugate gradient in the way that it stores

and updates the information that is accumulated. Instead of requiring intermediate storage on the order of N , the number of dimensions, it requires a matrix of size $N \times N$. Generally, for any moderate N , this is an entirely trivial disadvantage.

On the other hand, there is not, as far as we know, any overwhelming advantage that the variable metric methods hold over the conjugate gradient techniques, except perhaps a historical one. Developed somewhat earlier, and more widely propagated, the variable metric methods have by now developed a wider constituency of satisfied users. Likewise, some fancier implementations of variable metric methods (going beyond the scope of this book, see below) have been developed to a greater level of sophistication on issues like the minimization of roundoff error, handling of special conditions, and so on. We tend to use variable metric rather than conjugate gradient, but we have no reason to urge this habit on you.

Variable metric methods come in two main flavors. One is the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to as simply *Fletcher-Powell*). The other goes by the name *Broyden-Fletcher-Goldfarb-Shanno (BFGS)*. The BFGS and DFP schemes differ only in details of their roundoff error, convergence tolerances, and similar "dirty" issues which are outside of our scope [1,2]. However, it has become generally recognized that, empirically, the BFGS scheme is superior in these details. We will implement BFGS in this section.

As before, we imagine that our arbitrary function $f(\mathbf{x})$ can be locally approximated by the quadratic form of equation (10.6.1). We don't, however, have any information about the values of the quadratic form's parameters \mathbf{A} and \mathbf{b} , except insofar as we can glean such information from our function evaluations and line minimizations.

The basic idea of the variable metric method is to build up, iteratively, a good approximation to the inverse Hessian matrix \mathbf{A}^{-1} , that is, to construct a sequence of matrices \mathbf{H}_i with the property,

$$\lim_{i \rightarrow \infty} \mathbf{H}_i = \mathbf{A}^{-1} \quad (10.7.1)$$

Even better if the limit is achieved after N iterations instead of ∞ .

The reason that variable metric methods are sometimes called quasi-Newton methods can now be explained. Consider finding a minimum by using Newton's method to search for a zero of the gradient of the function. Near the current point \mathbf{x}_i , we have to second order

$$f(\mathbf{x}) = f(\mathbf{x}_i) + (\mathbf{x} - \mathbf{x}_i) \cdot \nabla f(\mathbf{x}_i) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_i) \cdot \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) \quad (10.7.2)$$

so

$$\nabla f(\mathbf{x}) = \nabla f(\mathbf{x}_i) + \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) \quad (10.7.3)$$

In Newton's method we set $\nabla f(\mathbf{x}) = 0$ to determine the next iteration point:

$$\mathbf{x} - \mathbf{x}_i = -\mathbf{A}^{-1} \cdot \nabla f(\mathbf{x}_i) \quad (10.7.4)$$

The left-hand side is the finite step we need take to get to the exact minimum; the right-hand side is known once we have accumulated an accurate $\mathbf{H} \approx \mathbf{A}^{-1}$.

The "quasi" in quasi-Newton is because we don't use the actual Hessian matrix of f , but instead use our current approximation of it. This is often *better* than

using the true Hessian. We can understand this paradoxical result by considering the *descent directions* of f at \mathbf{x}_i . These are the directions \mathbf{p} along which f decreases: $\nabla f \cdot \mathbf{p} < 0$. For the Newton direction (10.7.4) to be a descent direction, we must have

$$\nabla f(\mathbf{x}_i) \cdot (\mathbf{x} - \mathbf{x}_i) = -(\mathbf{x} - \mathbf{x}_i) \cdot \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) < 0 \quad (10.7.5)$$

that is, \mathbf{A} must be positive definite. In general, far from a minimum, we have no guarantee that the Hessian is positive definite. Taking the actual Newton step with the real Hessian can move us to points where the function is *increasing* in value. The idea behind quasi-Newton methods is to start with a positive definite, symmetric approximation to \mathbf{A} (usually the unit matrix) and build up the approximating \mathbf{H}_i 's in such a way that the matrix \mathbf{H}_i remains positive definite and symmetric. Far from the minimum, this guarantees that we always move in a downhill direction. Close to the minimum, the updating formula approaches the true Hessian and we enjoy the quadratic convergence of Newton's method.

When we are not close enough to the minimum, taking the full Newton step \mathbf{p} even with a positive definite \mathbf{A} need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that *initially* f decreases as we move in the Newton direction. Once again we can use the backtracking strategy described in §9.7 to choose a step along the *direction* of the Newton step \mathbf{p} , but not necessarily all the way.

We won't rigorously derive the DFP algorithm for taking \mathbf{H}_i into \mathbf{H}_{i+1} ; you can consult [3] for clear derivations. Following Brodlie (in [2]), we will give the following heuristic motivation of the procedure.

Subtracting equation (10.7.4) at \mathbf{x}_{i+1} from that same equation at \mathbf{x}_i gives

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{A}^{-1} \cdot (\nabla f_{i+1} - \nabla f_i) \quad (10.7.6)$$

where $\nabla f_j \equiv \nabla f(\mathbf{x}_j)$. Having made the step from \mathbf{x}_i to \mathbf{x}_{i+1} , we might reasonably want to require that the new approximation \mathbf{H}_{i+1} satisfy (10.7.6) as if it were actually \mathbf{A}^{-1} , that is,

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}_{i+1} \cdot (\nabla f_{i+1} - \nabla f_i) \quad (10.7.7)$$

We might also imagine that the updating formula should be of the form $\mathbf{H}_{i+1} = \mathbf{H}_i + \text{correction}$.

What "objects" are around out of which to construct a correction term? Most notable are the two vectors $\mathbf{x}_{i+1} - \mathbf{x}_i$ and $\nabla f_{i+1} - \nabla f_i$; and there is also \mathbf{H}_i . There are not infinitely many natural ways of making a matrix out of these objects, especially if (10.7.7) must hold! One such way, the *DFP updating formula*, is

$$\begin{aligned} \mathbf{H}_{i+1} = \mathbf{H}_i &+ \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i) \otimes (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} \\ &- \frac{[\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)] \otimes [\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \end{aligned} \quad (10.7.8)$$

where \otimes denotes the "outer" or "direct" product of two vectors, a matrix: The ij component of $\mathbf{u} \otimes \mathbf{v}$ is $u_i v_j$. (You might want to verify that 10.7.8 does satisfy 10.7.7.)

The BFGS updating formula is exactly the same, but with one additional term.

$$\dots + [(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)] \mathbf{u} \otimes \mathbf{u} \quad (10.7.9)$$

where \mathbf{u} is defined as the vector

$$\mathbf{u} \equiv \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} - \frac{\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \quad (10.7.10)$$

(You might also verify that this satisfies 10.7.7.)

You will have to take on faith — or else consult [3] for details of — the “deep” result that equation (10.7.8), with or without (10.7.9), does in fact converge to \mathbf{A}^{-1} in N steps, if f is a quadratic form.

Here now is the routine `dfpmin` that implements the quasi-Newton method, and uses `lnsrch` from §9.7. As mentioned at the end of `newt` in §9.7, this algorithm can fail if your variables are badly scaled.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 200
#define EPS 3.0e-8
#define TOLX (4*EPS)
#define STPMX 100.0

#define FREEALL free_vector(xi,1,n);free_vector(pnew,1,n); \
free_matrix(hessin,1,n,1,n);free_vector(hdg,1,n);free_vector(g,1,n); \
free_vector(dg,1,n);

void dfpmin(float p[], int n, float gtol, int *iter, float *fret,
float(*func)(float []), void (*dfunc)(float [], float []))
Given a starting point p[1..n] that is a vector of length n, the Broyden-Fletcher-Goldfarb-
Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function func, using
its gradient as calculated by a routine dfunc. The convergence requirement on zeroing the
gradient is input as gtol. Returned quantities are p[1..n] (the location of the minimum),
iter (the number of iterations that were performed), and fret (the minimum value of the
function). The routine lnsrch is called to perform approximate line minimizations.
{
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
float *f, float stpmax, int *check, float (*func)(float []));
    int check,i,its,j;
    float den,fac,fad,fae,fp,stpmax,sum=0.0,sumdg,sumxi,temp,test;
    float *dg,*g,*hdg,**hessin,*pnew,*xi;

    dg=vector(1,n);
    g=vector(1,n);
    hdg=vector(1,n);
    hessin=matrix(1,n,1,n);
    pnew=vector(1,n);
    xi=vector(1,n);
    fp=(*func)(p);
    (*dfunc)(p,g);
    for (i=1;i<=n;i++) {
        for (j=1;j<=n;j++) hessin[i][j]=0.0;
        hessin[i][i]=1.0;
        xi[i] = -g[i];
        sum += p[i]*p[i];
    }
    stpmax=STPMX*FMAX(sqrt(sum),(float)n);
```

Maximum allowed number of iterations.
Machine precision.
Convergence criterion on x values.
Scaled maximum step length allowed in
line searches.

Calculate starting function value and gra-
dient,
and initialize the inverse Hessian to the
unit matrix.

Initial line direction.

one additional term,

(10.7.9)

(10.7.10)

ls of — the “deep”
t converge to A^{-1}

ewton method, and
.7, this algorithm

umber of iterations.

n on x values.
p length allowed in

1,n); \

Fletcher-Goldfarb-
nction func, using
nt on zeroing the
of the minimum),
num value of the
ations.

p[], float x[],
[]));

n value and gra-

Hessian to the

```

for (its=1; its<=ITMAX; its++) {           Main loop over the iterations.
    *iter=its;
    lnsrch(n,p,fp,g,xi,pnew,fret,stpmax,&check,func);
    The new function evaluation occurs in lnsrch; save the function value in fp for the
    next line search. It is usually safe to ignore the value of check.
    fp = *fret;
    for (i=1; i<=n; i++) {
        xi[i]=pnew[i]-p[i];
        p[i]=pnew[i];
        Update the line direction,
        and the current point.
    }
    test=0.0;
    Test for convergence on  $\Delta x$ .
    for (i=1; i<=n; i++) {
        temp=fabs(xi[i])/FMAX(fabs(p[i]),1.0);
        if (temp > test) test=temp;
    }
    if (test < TOLX) {
        FREEALL
        return;
    }
    for (i=1; i<=n; i++) dg[i]=g[i];
    (*dfunc)(p,g);
    Save the old gradient,
    and get the new gradient.
    test=0.0;
    den=FMAX(*fret,1.0);
    Test for convergence on zero gradient.
    for (i=1; i<=n; i++) {
        temp=fabs(g[i])*FMAX(fabs(p[i]),1.0)/den;
        if (temp > test) test=temp;
    }
    if (test < gtol) {
        FREEALL
        return;
    }
    for (i=1; i<=n; i++) dg[i]=g[i]-dg[i];
    Compute difference of gradients,
    and difference times current matrix.
    for (i=1; i<=n; i++) {
        hdg[i]=0.0;
        for (j=1; j<=n; j++) hdg[i] += hessin[i][j]*dg[j];
    }
    fac=fae=sumdg=sumxi=0.0;
    Calculate dot products for the denomi-
    nators.
    for (i=1; i<=n; i++) {
        fac += dg[i]*xi[i];
        fae += dg[i]*hdg[i];
        sumdg += SQR(dg[i]);
        sumxi += SQR(xi[i]);
    }
    if (fac > sqrt(EPS*sumdg*sumxi)) {
        Skip update if fac not sufficiently posi-
        tive.
        fac=1.0/fac;
        fad=1.0/fae;
        The vector that makes BFGS different from DFP:
        for (i=1; i<=n; i++) dg[i]=fac*xi[i]-fad*hdg[i];
        for (i=1; i<=n; i++) {
            The BFGS updating formula:
            for (j=i; j<=n; j++) {
                hessin[i][j] += fac*xi[i]*xi[j]
                -fad*hdg[i]*hdg[j]+fae*dg[i]*dg[j];
                hessin[j][i]=hessin[i][j];
            }
        }
    }
    for (i=1; i<=n; i++) {
        xi[i]=0.0;
        Now calculate the next direction to go.
        for (j=1; j<=n; j++) xi[i] -= hessin[i][j]*g[j];
    }
    and go back for another iteration.
}
nrerror("too many iterations in dfpmin");
FREEALL
}

```

Quasi-Newton methods like `dfpmin` work well with the approximate line minimization done by `lnsrch`. The routines `powell` (§10.5) and `frprmn` (§10.6), however, need more accurate line minimization, which is carried out by the routine `linmin`.

Advanced Implementations of Variable Metric Methods

Although rare, it can conceivably happen that roundoff errors cause the matrix H_i to become nearly singular or non-positive-definite. This can be serious, because the supposed search directions might then not lead downhill, and because nearly singular H_i 's tend to give subsequent H_i 's that are also nearly singular.

There is a simple fix for this rare problem, the same as was mentioned in §10.4: In case of any doubt, you should *restart* the algorithm at the claimed minimum point, and see if it goes anywhere. Simple, but not very elegant. Modern implementations of variable metric methods deal with the problem in a more sophisticated way.

Instead of building up an approximation to A^{-1} , it is possible to build up an approximation of A itself. Then, instead of calculating the left-hand side of (10.7.4) directly, one solves the set of linear equations

$$A \cdot (x_m - x_i) = -\nabla f(x_i) \quad (10.7.11)$$

At first glance this seems like a bad idea, since solving (10.7.11) is a process of order N^3 — and anyway, how does this help the roundoff problem? The trick is not to store A but rather a triangular decomposition of A , its *Cholesky decomposition* (cf. §2.9). The updating formula used for the Cholesky decomposition of A is of order N^2 and can be arranged to guarantee that the matrix remains positive definite and nonsingular, even in the presence of finite roundoff. This method is due to Gill and Murray [1,2].

CITED REFERENCES AND FURTHER READING:

- Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]
 Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1, §§3–6 (by K. W. Brodlie). [2]
 Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), pp. 56ff. [3]
 Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 467–468.

10.8 Linear Programming and the Simplex Method

The subject of *linear programming*, sometimes called *linear optimization*, concerns itself with the following problem: For N independent variables x_1, \dots, x_N , maximize the function

$$z = a_{01}x_1 + a_{02}x_2 + \dots + a_{0N}x_N \quad (10.8.1)$$

subject to the primary constraints

$$x_1 \geq 0, \quad x_2 \geq 0, \quad \dots \quad x_N \geq 0 \quad (10.8.2)$$

2

Thus our strategy is quite simple: We always first try the full Newton step, because once we are close enough to the solution we will get quadratic convergence. However, we check at each iteration that the proposed step reduces f . If not, we *backtrack* along the Newton direction until we have an acceptable step. Because the Newton step is a descent direction for f , we are guaranteed to find an acceptable step by backtracking. We will discuss the backtracking algorithm in more detail below.

Note that this method essentially minimizes f by taking Newton steps designed to bring \mathbf{F} to zero. This is *not* equivalent to minimizing f directly by taking Newton steps designed to bring ∇f to zero. While the method can still occasionally fail by landing on a local minimum of f , this is quite rare in practice. The routine `newt` below will warn you if this happens. The remedy is to try a new starting point.

Line Searches and Backtracking

When we are not close enough to the minimum of f , taking the full Newton step $\mathbf{p} = \delta \mathbf{x}$ need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that *initially* f decreases as we move in the Newton direction. So the goal is to move to a new point \mathbf{x}_{new} along the *direction* of the Newton step \mathbf{p} , but not necessarily all the way:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \lambda \mathbf{p}, \quad 0 < \lambda \leq 1 \quad (9.7.6)$$

The aim is to find λ so that $f(\mathbf{x}_{\text{old}} + \lambda \mathbf{p})$ has decreased sufficiently. Until the early 1970s, standard practice was to choose λ so that \mathbf{x}_{new} exactly minimizes f in the direction \mathbf{p} . However, we now know that it is extremely wasteful of function evaluations to do so. A better strategy is as follows: Since \mathbf{p} is always the Newton direction in our algorithms, we first try $\lambda = 1$, the full Newton step. This will lead to quadratic convergence when \mathbf{x} is sufficiently close to the solution. However, if $f(\mathbf{x}_{\text{new}})$ does not meet our acceptance criteria, we *backtrack* along the Newton direction, trying a smaller value of λ , until we find a suitable point. Since the Newton direction is a descent direction, we are guaranteed to decrease f for sufficiently small λ .

What should the criterion for accepting a step be? It is *not* sufficient to require merely that $f(\mathbf{x}_{\text{new}}) < f(\mathbf{x}_{\text{old}})$. This criterion can fail to converge to a minimum of f in one of two ways. First, it is possible to construct a sequence of steps satisfying this criterion with f decreasing too slowly relative to the step lengths. Second, one can have a sequence where the step lengths are too small relative to the initial rate of decrease of f . (For examples of such sequences, see [1], p. 117.)

A simple way to fix the first problem is to require the *average* rate of decrease of f to be at least some fraction α of the *initial* rate of decrease $\nabla f \cdot \mathbf{p}$:

$$f(\mathbf{x}_{\text{new}}) \leq f(\mathbf{x}_{\text{old}}) + \alpha \nabla f \cdot (\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{old}}) \quad (9.7.7)$$

Here the parameter α satisfies $0 < \alpha < 1$. We can get away with quite small values of α ; $\alpha = 10^{-4}$ is a good choice.

The second problem can be fixed by requiring the rate of decrease of f at \mathbf{x}_{new} to be greater than some fraction β of the rate of decrease of f at \mathbf{x}_{old} . In practice, we will not need to impose this second constraint because our backtracking algorithm will have a built-in cutoff to avoid taking steps that are too small.

Here is the strategy for a practical backtracking routine: Define

$$g(\lambda) \equiv f(\mathbf{x}_{\text{old}} + \lambda \mathbf{p}) \quad (9.7.8)$$

so that

$$g'(\lambda) = \nabla f \cdot \mathbf{p} \quad (9.7.9)$$

If we need to backtrack, then we model g with the most current information we have and choose λ to minimize the model. We start with $g(0)$ and $g'(0)$ available. The first step is

always the Newton step, $\lambda = 1$. If this step is not acceptable, we have available $g(1)$ as well. We can therefore model $g(\lambda)$ as a quadratic:

$$g(\lambda) \approx [g(1) - g(0) - g'(0)]\lambda^2 + g'(0)\lambda + g(0) \quad (9.7.10)$$

Taking the derivative of this quadratic, we find that it is a minimum when

$$\lambda = -\frac{g'(0)}{2[g(1) - g(0) - g'(0)]} \quad (9.7.11)$$

Since the Newton step failed, we can show that $\lambda \lesssim \frac{1}{2}$ for small α . We need to guard against too small a value of λ , however. We set $\lambda_{\min} = 0.1$.

On second and subsequent backtracks, we model g as a cubic in λ , using the previous value $g(\lambda_1)$ and the second most recent value $g(\lambda_2)$:

$$g(\lambda) = a\lambda^3 + b\lambda^2 + g'(0)\lambda + g(0) \quad (9.7.12)$$

Requiring this expression to give the correct values of g at λ_1 and λ_2 gives two equations that can be solved for the coefficients a and b :

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} 1/\lambda_1^2 & -1/\lambda_2^2 \\ -\lambda_2/\lambda_1^2 & \lambda_1/\lambda_2^2 \end{bmatrix} \cdot \begin{bmatrix} g(\lambda_1) - g'(0)\lambda_1 - g(0) \\ g(\lambda_2) - g'(0)\lambda_2 - g(0) \end{bmatrix} \quad (9.7.13)$$

The minimum of the cubic (9.7.12) is at

$$\lambda = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a} \quad (9.7.14)$$

We enforce that λ lie between $\lambda_{\max} = 0.5\lambda_1$ and $\lambda_{\min} = 0.1\lambda_1$.

The routine has two additional features, a minimum step length `alamin` and a maximum step length `stpmax`. `lnsrch` will also be used in the quasi-Newton minimization routine `dfpmin` in the next section.

```
#include <math.h>
#include "nrutil.h"
#define ALF 1.0e-4
#define TOLX 1.0e-7
```

Ensures sufficient decrease in function value.
Convergence criterion on Δx .

```
void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
float *f, float stpmax, int *check, float (*func)(float []))
Given an n-dimensional point xold[1..n], the value of the function and gradient there, fold
and g[1..n], and a direction p[1..n], finds a new point x[1..n] along the direction p from
xold where the function func has decreased "sufficiently." The new function value is returned
in f. stpmax is an input quantity that limits the length of the steps so that you do not try to
evaluate the function in regions where it is undefined or subject to overflow. p is usually the
Newton direction. The output quantity check is false (0) on a normal exit. It is true (1) when
x is too close to xold. In a minimization algorithm, this usually signals convergence and can
be ignored. However, in a zero-finding algorithm the calling program should check whether the
convergence is spurious. Some "difficult" problems may require double precision in this routine.
```

```
{
int i;
float a,alam,alam2,alamin,b,disc,f2,rhs1,rhs2,slope,sum,temp,
test,tmplam;
```

```
*check=0;
for (sum=0.0,i=1;i<=n;i++) sum += p[i]*p[i];
sum=sqrt(sum);
if (sum > stpmax)
for (i=1;i<=n;i++) p[i] *= stpmax/sum; Scale if attempted step is too big.
for (slope=0.0,i=1;i<=n;i++)
slope += g[i]*p[i];
if (slope >= 0.0) nrerror("Roundoff problem in lnsrch.");
test=0.0; Compute  $\lambda_{\min}$ .
for (i=1;i<=n;i++) {
```

(9.7.6)

(9.7.7)

(9.7.8)

(9.7.9)

```

    temp=fabs(p[i])/FMAX(fabs(xold[i]),1.0);
    if (temp > test) test=temp;
}
alamin=TOLX/test;
alam=1.0;
for (;;) {
    for (i=1;i<=n;i++) x[i]=xold[i]+alam*p[i];
    *f=(*func)(x);
    if (alam < alamin) {
        for (i=1;i<=n;i++) x[i]=xold[i];
        *check=1;
        return;
    } else if (*f <= fold+ALF*alam*slope) return;
    else {
        if (alam == 1.0)
            tmplam = -slope/(2.0*(f-fold-slope));
        else {
            rhs1 = *f-fold-alam*slope;
            rhs2=f2-fold-alam2*slope;
            a=(rhs1/(alam*alam)-rhs2/(alam2*alam2))/(alam-alam2);
            b=(-alam2*rhs1/(alam*alam)+alam*rhs2/(alam2*alam2))/(alam-alam2);
            if (a == 0.0) tmplam = -slope/(2.0*b);
            else {
                disc=b*b-3.0*a*slope;
                if (disc < 0.0) tmplam=0.5*alam;
                else if (b <= 0.0) tmplam=(-b+sqrt(disc))/(3.0*a);
                else tmplam=-slope/(b+sqrt(disc));
            }
            if (tmplam > 0.5*alam)
                tmplam=0.5*alam;
        }
    }
    alam2=alam;
    f2 = *f;
    alam=FMAX(tmplam,0.1*alam);
}

```

Always try full Newton step first.
Start of iteration loop.

Convergence on Δx . For zero finding, the calling program should verify the convergence.

Sufficient function decrease. Backtrack.

First time.
Subsequent backtracks.

$\lambda \leq 0.5\lambda_1$.

$\lambda \geq 0.1\lambda_1$.
Try again.

Here now is the globally convergent Newton routine `newt` that uses `lnsrch`. A feature of `newt` is that you need not supply the Jacobian matrix analytically; the routine will attempt to compute the necessary partial derivatives of `F` by finite differences in the routine `fdjac`. This routine uses some of the techniques described in §5.7 for computing numerical derivatives. Of course, you can always replace `fdjac` with a routine that calculates the Jacobian analytically if this is easy for you to do.

```

#include <math.h>
#include "nutil.h"
#define MAXITS 200
#define TOLF 1.0e-4
#define TOLMIN 1.0e-6
#define TOLX 1.0e-7
#define STPMX 100.0

```

Here `MAXITS` is the maximum number of iterations; `TOLF` sets the convergence criterion on function values; `TOLMIN` sets the criterion for deciding whether spurious convergence to a minimum of `fmin` has occurred; `TOLX` is the convergence criterion on δx ; `STPMX` is the scaled maximum step length allowed in line searches.

```

int nn;
float *fvec;
void (*ffuncv)(int n, float v[], float f[]);
#define FREERETURN {free_vector(fvec,1,n);free_vector(xold,1,n);\
    free_vector(p,1,n);free_vector(g,1,n);free_matrix(fjac,1,n,1,n);\
    free_ivec(xndx,1,n);return;}

```

Global variables to communicate with `fmin`.